Laravel 9 · База данных · Построитель запросов

Введение

Построитель запросов к базе данных Laravel предлагает удобный и гибкий интерфейс для создания и выполнения запросов к базе данных. Его можно использовать для выполнения большинства операций с базой данных в вашем приложении и он отлично работает со всеми поддерживаемыми Laravel системами баз данных.

Построитель запросов Laravel использует связывание параметров PDO для защиты приложения от SQL-инъекций. Нет необходимости чистить строки, передаваемые как связываемые параметры.

> {note} PDO не поддерживает связывание имен столбцов. Поэтому, вы никогда не должны использовать какие-либо входящие от пользователя данные в качестве имен столбцов, используемые вашими запросами, включая столбцы в запросах `order by` и т.д.

Выполнение запросов к базе данных

Получение всех строк из таблицы

Вы можете использовать метод `table` фасада `DB`, чтобы начать запрос. Метод `table` возвращает текучий экземпляр построителя запросов для данной таблицы, позволяя вам связать больше ограничений к запросу и, наконец, получить результаты, используя метод `get`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
```

```
{
    /**
     * Показать список всех пользователей приложения.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

Метод `get` возвращает экземпляр `Illuminate\Support\Collection`, содержащий результаты запроса, где каждый результат является экземпляром объекта `stdClass` PHP. Вы можете получить доступ к значению каждого столбца, обратившись к столбцу как к свойству объекта:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```

> {tip} Коллекции Laravel содержат множество чрезвычайно мощных методов для перебора и сокращения данных. Для получения дополнительной информации о коллекциях Laravel ознакомьтесь с их документацией.

Получение одной строки / столбца из таблицы

Если вам просто нужно получить одну строку из таблицы базы данных, вы можете использовать метод `first` фасада `DB`. Этот метод вернет единственный объект `stdClass`:

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

Если вам не нужна вся строка, вы можете извлечь одно значение из записи с помощью метода `value`. Этот метод вернет значение столбца напрямую:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Чтобы получить одну строку по значению столбца `id`, используйте метод `find`:

```
$user = DB::table('users')->find(3);
```

Получение списка значений столбца

Если вы хотите получить экземпляр `Illuminate\Support\Collection`, содержащий значения одного столбца, вы можете использовать метод `pluck`. В этом примере мы получим коллекцию из названий ролей:

```
use Illuminate\Support\Facades\DB;

$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

Вы можете указать столбец, который результирующая коллекция должна использовать в качестве ключей, указав второй аргумент методу `pluck`:

```
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
    echo $title;
}
```

Разбиение результатов

Если вам нужно работать с тысячами записей базы данных, рассмотрите возможность использования метода `chunk` фасада `DB`. Этот метод извлекает за раз небольшой фрагмент результатов и передает каждый фрагмент в замыкание для обработки. Например, давайте извлечем всю таблицу `users` фрагментами по 100 записей за раз:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

Вы можете остановить обработку последующих фрагментов, вернув из замыкания `false`:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // Обрабатываем записи ...

    return false;
});
```

Если вы обновляете записи базы данных во время фрагментирования результатов, то результаты ваших фрагментов могут измениться неожиданным образом. Если вы планируете обновлять полученные записи при фрагментировании, всегда лучше использовать вместо этого метод `chunkById`. Этот метод автоматически разбивает результаты на фрагменты на основе первичного ключа записи:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function ($users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]);
        }
    });
```

> {note} При обновлении или удалении записей внутри замыкания, любые изменения первичного или внешних ключей могут повлиять на запрос очередного фрагмента. Это может потенциально привести к тому, что записи могут не быть включены в последующие результаты выполнения замыкания.

Отложенная потоковая передача результатов

Метод `lazy` работает аналогично методу `chunk` в том смысле, что он выполняет запрос по частям. Однако вместо передачи каждого фрагмента непосредственно в замыкание, метод `lazy()` возвращает экземпляр `LazyCollection`, что позволяет вам взаимодействовать с результатами как с единым потоком:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->lazy()->each(function ($user) {
    //
});
```

Еще раз, если вы планируете обновлять полученные записи во время их итерации, лучше вместо этого использовать методы `lazyById` / `lazyByIdDesc`. Эти методы автоматически разбивает результаты «постранично» на основе первичного ключа записи:

```
DB::table('users')->where('active', false)
    ->lazyById()->each(function ($user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    });
```

> {note} При обновлении или удалении записей во время их итерации любые изменения первичного ключа или внешних ключей могут повлиять на запрос фрагмента. Это может потенциально привести к тому, что записи не будут включены в результирующий набор.

Извлечение Агрегатов

Построитель запросов также содержит множество методов для получения агрегированных значений, таких как `count`, `max`, `min`, `avg` и `sum`. После создания запроса вы можете вызвать любой из этих методов:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Конечно, вы можете комбинировать эти методы с другими выражениями, чтобы уточнить способ вычисления вашего совокупного значения:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Определение наличия записей

Вместо использования метода `count` для определения существования каких-либо записей, соответствующих ограничениям вашего запроса, используйте методы `exists` и `doesntExist`:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

Выражения Select

Уточнения выражения Select

Возможно, вам не всегда нужно выбирать все столбцы из таблицы базы данных. Используя метод `select`, вы можете указать собственное выражение `SELECT` для запроса:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->select('name', 'email as user_email')
            ->get();
```

Метод `distinct` позволяет вам заставить запрос возвращать уникальные результаты:

```
$users = DB::table('users')->distinct()->get();
```

Если у вас уже есть экземпляр построителя запросов, и вы хотите добавить столбец к существующему выражению `SELECT`, то вы можете использовать метод `addSelect`:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Необрабатываемые (сырые) выражения

Иногда требуется вставить в запрос произвольную строку. Чтобы создать необработанное строковое выражение, вы можете использовать метод `raw` фасада `DB`:

```
$users = DB::table('users')
             ->select(DB::raw('count(*) as user_count, status'))
             ->where('status', '<>', 1)
             ->groupBy('status')
             ->get();
```

> {note} Сырые выражения будут вставлены в запрос в виде строк, поэтому следует проявлять особую осторожность, чтобы не создавать уязвимости для SQL-инъекций.

Необрабатываемые методы

Вместо использования метода `DB::raw`, вы также можете использовать следующие методы для вставки необработанного выражения в различные части вашего запроса. **Помните, Laravel не может гарантировать, что любой запрос, использующий необработанные выражения, защищен от уязвимостей SQL-инъекций.**

`selectRaw`

Метод `selectRaw` можно использовать вместо `addSelect(DB::raw(/* ... */))`. Этот метод принимает необязательный массив связываемых параметров в качестве второго аргумента:

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as price_with_tax', [1.0825])
                ->get();
```

`whereRaw / orWhereRaw`

Методы `whereRaw` и `orWhereRaw` можно использовать для вставки необработанного выражения `WHERE` в ваш запрос. Эти методы принимают необязательный массив связываемых параметров в качестве второго аргумента:

```
$orders = DB::table('orders')
                ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
                ->get();
```

havingRaw / orHavingRaw

Методы `havingRaw` и `orHavingRaw` могут использоваться для вставки необработанной строки в качестве значения выражения `HAVING`. Эти методы принимают необязательный массив связываемых параметров в качестве второго аргумента:

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

orderByRaw

Метод `orderByRaw` используется для предоставления необработанной строки в качестве значения выражения `ORDER BY`:

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at DESC')
                ->get();
```

groupByRaw

Метод `groupByRaw` используется для предоставления необработанной строки в качестве значения выражения `GROUP BY`:

```
$orders = DB::table('orders')
                ->select('city', 'state')
                ->groupByRaw('city, state')
                ->get();
```

Соединения Joins

Inner Join

Построитель запросов также может использоваться для добавления выражений соединения к вашим запросам. Чтобы выполнить базовое «внутреннее соединение», вы можете использовать метод `join` экземпляра построителя. Первым аргументом, передаваемым методу `join`, является имя таблицы, к которой вам нужно присоединиться, а остальные аргументы определяют ограничения столбца для соединения. Вы даже можете соединить несколько таблиц в один запрос:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join / Right Join

Если вы хотите выполнить «левое соединение» или «правое соединение» вместо «внутреннего соединения», используйте методы `leftJoin` или `rightJoin`. Эти методы имеют ту же сигнатуру, что и метод `join`:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();

$users = DB::table('users')
            ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Cross Join

Вы можете использовать метод `crossJoin` для выполнения «перекрестного соединения». Перекрестные соединения генерируют декартово произведение между первой таблицей и соединяемой таблицей:

```
$sizes = DB::table('sizes')
            ->crossJoin('colors')
            ->get();
```

Расширенные выражения соединения

Вы также можете указать более сложные выражения соединения. Для начала передайте замыкание в качестве второго аргумента методу `join`. Замыкание получит экземпляр `Illuminate\Database\Query\JoinClause`, который позволяет вам указать ограничения `JOIN`:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(/* ... */);
        })
        ->get();
```

Если вы хотите использовать выражение `WHERE` в своих соединениях, вы можете использовать методы `where` и `orWhere` экземпляра `JoinClause`. Вместо сравнения двух столбцов эти методы будут сравнивать столбец со значением:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Подзапросы соединений

Вы можете использовать методы `joinSub`, `leftJoinSub` и `rightJoinSub`, чтобы присоединить запрос к подзапросу. Каждый из этих методов получает три аргумента: подзапрос, псевдоним таблицы и замыкание, определяющее связанные столбцы. В этом примере мы получим коллекцию пользователей, где каждая запись пользователя также содержит временную метку `created_at` последнего опубликованного поста пользователя в блоге:

```
$latestPosts = DB::table('posts')
                   ->select('user_id', DB::raw('MAX(created_at) as
last_post_created_at'))
                   ->where('is_published', true)
                   ->groupBy('user_id');

$users = DB::table('users')
        ->joinSub($latestPosts, 'latest_posts', function ($join) {
            $join->on('users.id', '=', 'latest_posts.user_id');
        })->get();
```

Объединения результатов Unions

Построитель запросов также содержит удобный метод «объединения» двух или более запросов вместе. Например, вы можете создать первый запрос и использовать метод `union` для объединения его с другими запросами:

```
use Illuminate\Support\Facades\DB;

$first = DB::table('users')
```

```
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

В дополнение к методу `union` , построитель запросов содержит метод `unionAll` . Запросы, объединенные с использованием метода `unionAll` , не будут удалять повторяющиеся результаты. Метод `unionAll` имеет ту же сигнатуру, что и метод `union` .

Основные выражения Where

Выражения Where

Вы можете использовать метод `where` построителя запросов, чтобы добавить в запрос выражения `WHERE` . Самый простой вызов метода `where` требует трех аргументов. Первый аргумент – это имя столбца. Второй аргумент – это оператор, который может быть любым из поддерживаемых базой данных операторов. Третий аргумент – это значение, которое нужно сравнить со значением столбца.

Например, следующий запрос извлекает пользователей, у которых значение столбца `votes` равно `100` , а значение столбца `age` больше, чем `35` :

```
$users = DB::table('users')
                ->where('votes', '=', 100)
                ->where('age', '>', 35)
                ->get();
```

Для удобства, если вы хотите убедиться, что столбец соответствует `=` переданному значению, то вы можете передать это значение в качестве второго аргумента в метод `where` . Laravel будет предполагать, что вы хотите использовать оператор `=` :

```
$users = DB::table('users')->where('votes', 100)->get();
```

Как упоминалось ранее, вы можете использовать любой оператор, который поддерживается вашей системой баз данных:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
```

```
                ->where('name', 'like', 'T%')
                ->get();
```

Вы также можете передать массив условий методу `where`. Каждый элемент массива должен быть массивом, содержащим три аргумента, как и обычно передаваемых методу `where`:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

> {note} PDO не поддерживает связывание имен столбцов. Поэтому, вы никогда не должны использовать какие-либо входящие от пользователя данные в качестве имен столбцов, используемые вашими запросами, включая столбцы в запросах `order by` и т.д.

Выражения Or Where

При объединении в цепочку вызовов метода `where` построителя запросов выражения `WHERE` будут объединены вместе с помощью оператора `AND`. Однако, вы можете использовать метод `orWhere` для добавления выражения к запросу с помощью оператора `OR`. Метод `orWhere` принимает те же аргументы, что и метод `where`:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

Если вам нужно сгруппировать условие `OR` в круглых скобках, вы можете передать замыкание в качестве первого аргумента методу `orWhere`:

```
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere(function($query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();
```

В приведенном выше примере будет получен следующий SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

> {note} Вы всегда должны группировать вызовы `orWhere`, чтобы избежать неожиданного поведения при применении глобальных областей запроса.

Выражения Where Not

Методы `whereNot` и `orWhereNot` могут использоваться для отмены переданной группы ограничений запроса. Например, следующий запрос исключает товары, которые находятся на распродаже или имеют цену меньше десяти:

```
$products = DB::table('products')
                ->whereNot(function ($query) {
                    $query->where('clearance', true)
                          ->orWhere('price', '<', 10);
                })
                ->get();
```

Выражения Where и JSON

Laravel также поддерживает запросы в базах данных, которые обеспечивают поддержку JSON-типов столбцов. В настоящее время это MySQL 5.7+, PostgreSQL, SQL Server 2016 и SQLite 3.9.0 (с расширением JSON1). Чтобы запросить столбец JSON, используйте оператор `->`:

```
$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

Вы можете использовать `whereJsonContains` для запроса массивов JSON. Эта функция не поддерживается базой данных SQLite:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', 'en')
                ->get();
```

Если ваше приложение использует базы данных MySQL или PostgreSQL, вы можете передать массив значений методу `whereJsonContains`:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', ['en', 'de'])
                ->get();
```

Вы можете использовать метод `whereJsonLength` для запроса массивов JSON по их длине:

```
$users = DB::table('users')
                ->whereJsonLength('options->languages', 0)
                ->get();

$users = DB::table('users')
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

Дополнительные выражения Where

whereBetween / orWhereBetween

Метод `whereBetween` проверяет, находится ли значение столбца между двумя значениями:

```
$users = DB::table('users')
           ->whereBetween('votes', [1, 100])
           ->get();
```

whereNotBetween / orWhereNotBetween

Метод `whereNotBetween` проверяет, что значение столбца находится вне двух значений:

```
$users = DB::table('users')
                    ->whereNotBetween('votes', [1, 100])
                    ->get();
```

whereIn / whereNotIn / orWhereIn / orWhereNotIn

Метод `whereIn` проверяет, что значение переданного столбца содержится в указанном массиве:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

Метод `whereNotIn` проверяет, что значение переданного столбца не содержится в указанном массиве:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

> {note} Если вы добавляете в свой запрос большой массив связываемых целочисленных параметров, то методы `whereIntegerInRaw` или `whereIntegerNotInRaw` могут использоваться для значительного сокращения потребляемой памяти.

whereNull / whereNotNull / orWhereNull / orWhereNotNull

Метод `whereNull` проверяет, что значение переданного столбца равно `NULL`:

```
$users = DB::table('users')
                ->whereNull('updated_at')
                ->get();
```

Метод `whereNotNull` проверяет, что значение переданного столбца не равно `NULL`:

```
$users = DB::table('users')
                ->whereNotNull('updated_at')
                ->get();
```

whereDate / whereMonth / whereDay / whereYear / whereTime

Метод `whereDate` используется для сравнения значения столбца с датой:

```
$users = DB::table('users')
                ->whereDate('created_at', '2016-12-31')
                ->get();
```

Метод `whereMonth` используется для сравнения значения столбца с конкретным месяцем:

```
$users = DB::table('users')
                ->whereMonth('created_at', '12')
                ->get();
```

Метод `whereDay` используется для сравнения значения столбца с определенным днем месяца:

```
$users = DB::table('users')
                ->whereDay('created_at', '31')
                ->get();
```

Метод `whereYear` используется для сравнения значения столбца с конкретным годом:

```
$users = DB::table('users')
                ->whereYear('created_at', '2016')
                ->get();
```

Метод `whereTime` используется для сравнения значения столбца с определенным временем:

```
$users = DB::table('users')
                ->whereTime('created_at', '=', '11:20:45')
                ->get();
```

whereColumn / orWhereColumn

Метод `whereColumn` используется для проверки равенства двух столбцов:

```
$users = DB::table('users')
                ->whereColumn('first_name', 'last_name')
                ->get();
```

Вы также можете передать оператор сравнения методу `whereColumn` :

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

Вы также можете передать массив сравнений столбцов методу `whereColumn`. Эти условия будут объединены с помощью оператора `AND`:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at'],
                ])->get();
```

Логическая группировка

Иногда требуется сгруппировать несколько выражений `WHERE` в круглых скобках, чтобы добиться желаемой логической группировки вашего запроса. Фактически, вы должны всегда группировать вызовы метода `orWhere` в круглых скобках, чтобы избежать неожиданного поведения запроса. Для этого вы можете передать замыкание методу `where`:

```
$users = DB::table('users')
           ->where('name', '=', 'John')
           ->where(function ($query) {
               $query->where('votes', '>', 100)
                     ->orWhere('title', '=', 'Admin');
           })
           ->get();
```

Передача замыкания в метод `where` указывает построителю запросов начать группу ограничений. Замыкание получит экземпляр построителя запросов, который вы можете использовать для задания ограничений, которые должны содержаться в группе скобок. В приведенном выше примере будет получен следующий SQL:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

> {note} Вы всегда должны группировать вызовы `orWhere`, чтобы избежать неожиданного поведения при применении глобальных областей запроса.

Расширенные выражения Where

Выражения Where Exists

Метод `whereExists` позволяет писать выражения `WHERE EXISTS` SQL. Метод `whereExists` принимает замыкание, которое получит экземпляр построителя запросов, позволяя вам определить запрос, который должен быть помещен внутри выражения `EXISTS`:

```
$users = DB::table('users')
           ->whereExists(function ($query) {
               $query->select(DB::raw(1))
                     ->from('orders')
                     ->whereColumn('orders.user_id', 'users.id');
           })
           ->get();
```

В приведенном выше примере будет получен следующий SQL:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

Подзапросы выражений Where

Иногда требуется создать выражение `WHERE`, которое сравнивает результаты подзапроса с переданным значением. Вы можете добиться этого, передав замыкание и значение методу `where`. Например, следующий запрос будет извлекать всех пользователей, недавно имевших «членство» указанного типа:

```
use App\Models\User;

$users = User::where(function ($query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

Иногда требуется создать выражение `WHERE`, которое сравнивает столбец с результатами подзапроса. Вы можете сделать это, передав столбец, оператор и замыкание методу `where`. Например, следующий запрос будет извлекать все записи о доходах, где сумма меньше средней:

```
use App\Models\Income;

$incomes = Income::where('amount', '<', function ($query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

Полнотекстовый поиск

> {note} Полнотекстовый поиск в настоящее время поддерживаются MySQL и PostgreSQL.

Методы `whereFullText` и `orWhereFullText` полнотекстового поиска можно использовать для добавления условий `where` в запрос для столбцов, которые имеют полнотекстовые индексы. Эти методы будут преобразованы Laravel в соответствующий SQL базы данных. Например, предложение `MATCH AGAINST` будет создано для приложений, использующих MySQL:

```
$users = DB::table('users')
           ->whereFullText('bio', 'web developer')
           ->get();
```

Сортировка, группировка, ограничение и смещение

Сортировка

Метод `orderBy`

Метод `orderBy` позволяет вам сортировать результаты запроса по конкретному столбцу. Первый аргумент, принимаемый методом `orderBy`, должен быть столбцом, по которому вы хотите выполнить сортировку, а второй аргумент определяет направление сортировки и может быть либо `asc`, либо `desc`:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

Для сортировки по нескольким столбцам вы можете просто вызывать `orderBy` столько раз, сколько необходимо:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->orderBy('email', 'asc')
                ->get();
```

Методы `latest` и `oldest`

Методы `latest` и `oldest` позволяют легко упорядочивать результаты по дате. По умолчанию результат будет упорядочен по столбцу `created_at` таблицы. Или вы можете передать имя столбца, по которому хотите сортировать:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

Случайный порядок

Метод `inRandomOrder` используется для случайной сортировки результатов запроса. Например, вы можете использовать этот метод для выборки случайного пользователя:

```
$randomUser = DB::table('users')
                ->inRandomOrder()
                ->first();
```

Удаление существующих сортировок

Метод `reorder` удаляет все выражения `ORDER BY`, которые ранее были применены к запросу:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

Вы можете передать столбец и направление при вызове метода `reorder`, чтобы удалить все существующие выражения `ORDER BY` и применить к запросу совершенно новый порядок:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

Группировка

Методы groupBy и having

Как и следовало ожидать, для группировки результатов запроса могут использоваться методы `groupBy` и `having`. Сигнатура метода `having` аналогична сигнатуре метода `where`:

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
                ->get();
```

Вы можете использовать метод `havingBetween` для фильтрации результатов по указанному диапазону:

```
$report = DB::table('orders')
                ->selectRaw('count(id) as number_of_orders, customer_id')
                ->groupBy('customer_id')
                ->havingBetween('number_of_orders', [5, 15])
                ->get();
```

Вы можете передать несколько аргументов методу `groupBy` для группировки по нескольким столбцам:

```
$users = DB::table('users')
                ->groupBy('first_name', 'status')
                ->having('account_id', '>', 100)
                ->get();
```

Чтобы создать более сложные операторы `having` , см. метод `havingRaw` .

Ограничение и смещение

Методы `skip` и `take`

Вы можете использовать методы `skip` и `take` , чтобы ограничить количество результатов, возвращаемых запросом, или пропустить указанное количество результатов из запроса:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Как вариант, вы можете использовать методы `limit` и `offset` . Эти методы функционально эквивалентны методам `take` и `skip` соответственно:

```
$users = DB::table('users')
                ->offset(10)
                ->limit(5)
                ->get();
```

Условные выражения

Иногда требуется, чтобы определенные выражения запроса применялись к запросу на основании другого условия. Например, бывает необходимо применить оператор `WHERE` только в том случае, если переданное входящее значение присутствует в HTTP-запросе. Вы можете сделать это с помощью метода `when` :

```
$role = $request->input('role');

$users = DB::table('users')
                ->when($role, function ($query, $role) {
                    $query->where('role_id', $role);
                })
                ->get();
```

Метод `when` выполняет переданное замыкание только тогда, когда первый аргумент равен `true`. Если первый аргумент – `false`, замыкание не будет выполнено. Итак, в приведенном выше примере замыкание метода `when` будет вызываться только в том случае, если поле `role` присутствует во входящем запросе и оценивается как `true`.

Вы можете передать другое замыкание в качестве третьего аргумента методу `when`. Это замыкание будет выполнено только в том случае, если первый аргумент оценивается как `false`. Чтобы проиллюстрировать этот функционал, мы будем использовать его для определения порядка вывода записей по умолчанию для запроса:

```
$sortByVotes = $request->input('sort_by_votes');

$users = DB::table('users')
                ->when($sortByVotes, function ($query, $sortByVotes) {
                    $query->orderBy('votes');
                }, function ($query) {
                    $query->orderBy('name');
                })
                ->get();
```

Вставка

Построитель запросов также содержит метод `insert`, который можно использовать для вставки записей в таблицу базы данных. Метод `insert` принимает массив имен и значений столбцов:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

Вы можете вставить сразу несколько записей, передав массив массивов. Каждый из массивов представляет собой запись, которую нужно вставить в таблицу:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

Метод `insertOrIgnore` будет игнорировать ошибки при вставке записей в базу данных. При использовании этого метода вы должны знать, что ошибки повторяющихся записей будут игнорироваться, а также (в зависимости от движка базы данных) могут быть проигнорированы другие типы ошибок. Например, `insertOrIgnore` будет обходить строгий режим MySQL:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

Метод `insertUsing` будет вставлять новые записи в таблицу, используя подзапрос для определения данных, которые должны быть вставлены:

```
DB::table('pruned_users')->insertUsing([
    'id', 'name', 'email', 'email_verified_at'
], DB::table('users')->select(
    'id', 'name', 'email', 'email_verified_at'
)->where('updated_at', '<=', now()->subMonth()));
```

Автоинкрементирование идентификаторов

Если таблица имеет автоинкрементный идентификатор, то используйте метод `insertGetId`, чтобы вставить запись и затем получить идентификатор этой записи:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

> {note} При использовании PostgreSQL метод `insertGetId` ожидает, что автоинкрементный столбец будет называться `id`. Если вы хотите получить идентификатор из другой «последовательности», вы можете передать имя столбца в качестве второго параметра методу `insertGetId`.

Обновления-вставки

Метод `upsert` вставляет записи, которые не существуют, и обновляет записи, которые уже существуют, новыми значениями, которые вы можете указать. Первый аргумент метода состоит из значений для вставки или обновления, а второй аргумент перечисляет столбцы, которые однозначно идентифицируют записи в связанной таблице. Третий и последний аргумент метода – это массив столбцов, который следует обновить, если соответствующая запись уже существует в базе данных:

```
DB::table('flights')->upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

В приведенном выше примере Laravel попытается вставить две записи. Если запись уже существует с такими же значениями столбцов `departure` и `destination`, то Laravel обновит столбец `price` этой записи.

> {note} Все базы данных, кроме SQL Server, требуют, чтобы столбцы во втором аргументе метода `upsert` имели «первичный» или «уникальный» индекс. Вдобавок, драйвер базы данных MySQL игнорирует второй аргумент метода `upsert` и всегда использует «первичный» и «уникальный» индексы таблицы для обнаружения существующих записей.

Обновление

Помимо вставки записей в базу данных, построитель запросов также может обновлять существующие записи с помощью метода `update`. Метод `update`, как и метод `insert`, принимает массив пар столбцов и значений, указывающих столбцы, которые нужно обновить. Метод `update` возвращает количество затронутых строк. Вы можете ограничить запрос `update` с помощью выражений `WHERE`:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['votes' => 1]);
```

Обновление или вставка

Иногда требуется обновить существующую запись в базе данных или создать ее, если соответствующей записи не существует. В этом сценарии может использоваться метод `updateOrInsert`. Метод `updateOrInsert` принимает два аргумента: массив условий, по которым нужно найти запись, и массив пар столбцов и значений, указывающих столбцы, которые нужно обновить.

Метод `updateOrInsert` попытается найти соответствующую запись в базе данных, используя пары столбец и значение первого аргумента. Если запись существует, она будет обновлена значениями второго аргумента. Если запись не может быть найдена, будет вставлена новая запись с объединенными атрибутами обоих аргументов:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

Обновление столбцов JSON

При обновлении столбца JSON вы должны использовать синтаксис `->` для обновления соответствующего ключа в объекте JSON. Эта операция поддерживается в MySQL 5.7+ и PostgreSQL 9.5+:

```
$affected = DB::table('users')
              ->where('id', 1)
```

```
    ->update(['options->enabled' => true]);
```

Увеличение и уменьшение отдельных значений

Конструктор запросов также содержит удобные методы увеличения или уменьшения значения конкретного столбца. Оба эти метода принимают по крайней мере один аргумент: столбец, который нужно изменить. Может быть указан второй аргумент, определяющий величину, на которую следует увеличить или уменьшить столбец:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

Вы также можете указать дополнительные столбцы для обновления во время выполнения запроса:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Удаление

Метод `delete` построителя запросов может использоваться для удаления записей из таблицы. Метод `delete` возвращает количество затронутых строк. Вы можете ограничить операторы `delete` , добавив метод `where` перед вызовом метода `delete` :

```
$deleted =DB::table('users')->delete();

$deleted =DB::table('users')->where('votes', '>', 100)->delete();
```

Если вы хотите очистить всю таблицу, что приведет к удалению всех записей из таблицы и сбросу автоинкрементного идентификатора на ноль, вы можете использовать метод `truncate` :

```
DB::table('users')->truncate();
```

Очистка таблицы и PostgreSQL

При очистке базы данных PostgreSQL будет применено поведение `CASCADE` . Это означает, что все связанные с внешним ключом записи в других таблицах также будут удалены.

Пессимистическая блокировка

Построитель запросов также включает несколько функций, которые помогут вам достичь «пессимистической блокировки» при выполнении ваших операторов `SELECT`. Чтобы выполнить оператор с «совместной блокировкой», вы можете вызвать метод `sharedLock` в запросе. Совместная блокировка предотвращает изменение выбранных строк до тех пор, пока ваша транзакция не будет зафиксирована:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->sharedLock()
        ->get();
```

В качестве альтернативы вы можете использовать метод `lockForUpdate`. Блокировка «для обновления» предотвращает изменение выбранных записей или их выбор с помощью другой совместной блокировки:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->lockForUpdate()
        ->get();
```

Отладка

Вы можете использовать методы `dd` или `dump` при построении запроса, чтобы отобразить связанные параметры запроса и сам SQL-запрос. Метод `dd` отобразит отладочную информацию и затем прекратит выполнение запроса. Метод `dump` отобразит информацию об отладке, но позволит продолжить выполнение запроса:

```
DB::table('users')->where('votes', '>', 100)->dd();

DB::table('users')->where('votes', '>', 100)->dump();
```